🛠️ Ejercicio Práctico

📌 **Título:** *Cifrado de Contraseñas y Almacenamiento Seguro*

🎯 Objetivo del ejercicio:

Implementar un sistema básico de **cifrado de contraseñas** utilizando un algoritmo de **hashing** y **salting**, asegurando que las contraseñas de los usuarios sean almacenadas de manera segura en la base de datos.

📝 Escenario:

Estás desarrollando una **aplicación web** para gestionar cuentas de usuarios. Para proteger las contraseñas de los usuarios, necesitas implementar un sistema que utilice **hashing** y **salting** para almacenar las contraseñas de forma segura.

🔍 Tu tarea:

Paso 1 – Instalar librerías necesarias:
Instala **bcrypt** en tu proyecto de **Node.js** (o la librería equivalente en el lenguaje que prefieras) para poder realizar el **hashing** y **salting** de las contraseñas. Si usas **Node.js**, el comando es:

```
npm install bcrypt
```

1.

Paso 2 – Crear una función para almacenar contraseñas:

1. Crea una **función** que permita almacenar la contraseña de un usuario de manera segura:
    - La contraseña debe ser **hasheada** utilizando el algoritmo **bcrypt**.

- La función debe **añadir un "sal" aleatorio** a la contraseña antes de aplicar el **hashing**.

- La función debe retornar el **hash** resultante.

Ejemplo de implementación (Node.js):

```
const bcrypt = require('bcrypt');
const saltRounds = 10;

const almacenarContraseña = async (contraseña) => {
  try {
    // Generar un sal aleatorio
    const salt = await bcrypt.genSalt(saltRounds);
    // Hash de la contraseña con el sal
    const hash = await bcrypt.hash(contraseña, salt);
    return hash;
  } catch (err) {
    console.error("Error al almacenar la contraseña:", err);
  }
};
```

Paso 3 – Verificar las contraseñas al inicio de sesión:

1. Crea una función que permita **verificar** si la contraseña proporcionada por el usuario coincide con el **hash** almacenado:

    - La función debe usar **bcrypt.compare()** para comparar la contraseña proporcionada con el **hash** almacenado.

Ejemplo de implementación (Node.js):

```
const verificarContraseña = async (contraseñaProporcionada, hashAlmacenado) => {
  try {
    const esCorrecta = await bcrypt.compare(contraseñaProporcionada, hashAlmacenado);
    if (esCorrecta) {
      console.log("Contraseña correcta.");
    } else {
      console.log("Contraseña incorrecta.");
    }
  } catch (err) {
    console.error("Error al verificar la contraseña:", err);
  }
};
```

Paso 4 – Probar el sistema de cifrado de contraseñas:

1. Registra una nueva contraseña utilizando la función **almacenarContraseña()**.

2. Luego, al intentar iniciar sesión, compara la contraseña proporcionada con el **hash** almacenado utilizando la función **verificarContraseña()**.

Ejemplo de uso:

```
// Simulando el registro de una nueva contraseña
const contraseña = "miContraseñaSegura123";
almacenarContraseña(contraseña)
  .then((hash) => {
    console.log("Contraseña almacenada (hash):", hash);

    // Simulando el inicio de sesión y verificación de la contraseña
    verificarContraseña("miContraseñaSegura123", hash);  // Contraseña correcta
    verificarContraseña("incorrecta123", hash);  // Contraseña incorrecta
  });
```

✅ Resultado esperado:

- Un sistema funcional donde las contraseñas de los usuarios se almacenan de manera segura utilizando **bcrypt** para hashing y salting.

- Las contraseñas no deben ser almacenadas en texto claro, sino como **hashes** generados con **salting**.

- Una función de **verificación de contraseñas** que valide correctamente la contraseña proporcionada al inicio de sesión contra el hash almacenado.

📄 Entrega sugerida:

1. **Código fuente** de la implementación de las funciones de **almacenamiento** y **verificación de contraseñas**.

2. **Capturas de pantalla** o una pequeña demostración en consola mostrando la **almacenación** y la **verificación** de las contraseñas.

3. **Informe breve** que explique cómo se implementó el **hashing** y el **salting** de contraseñas.

🧰 Herramientas recomendadas:

- **Node.js** con **bcrypt** (o el equivalente en tu lenguaje preferido).
- **Postman** o **Insomnia** para probar las rutas de la API si es necesario.
- **MongoDB** o **MySQL** para almacenar los **hashes** de las contraseñas de los usuarios.

📝 Sugerencia para extender el ejercicio (opcional):

- **Autenticación Multifactor (MFA)**: Implementa un sistema de autenticación adicional que requiera un segundo factor (como un código enviado por correo electrónico o SMS).
- **Token de actualización (Refresh Token)**: Añade la posibilidad de usar un **refresh token** para mantener la sesión del usuario activa después de que el JWT haya expirado.
